Inductive and Coinductive Data Types in Typed Lambda Calculus Revisited

Herman Geuvers

Radboud University Nijmegen
and
Eindhoven University of Technology
The Netherlands

TLCA 2015, Warsaw, July 2

Contents

How a programmer may look at a recursive function

P: Given $f : A \to A$, I want to loop f until it stops
T: But if you keep calling f, it will never stop
P: Ehh...
T: You mean that you have a function $f : A \to A + B$, and if you get a value in A, you continue, and if you get a value in B, you stop?
P: That's right! And the function I want to define in the end is from A to B anyway!
T: So you want

$$\frac{f : A \to A + B}{\mathrm{loop}\, f : A \to B}$$

satisfying

$$\mathrm{loop}\, f\ a = \mathrm{case}\ f\ a\ \mathrm{of}\ (\mathrm{inl}\, a' \Rightarrow \mathrm{loop}\, f\ a')(\mathrm{inr}\, b \Rightarrow b)$$

Can we dualize this looping?

$$\frac{f : A \to A + B}{\mathrm{loop}\, f : A \to B}$$

$$\mathrm{loop}\, f\ a = \mathrm{case}\ f\ a\ \mathrm{of}\ (\mathrm{inl}\, a' \Rightarrow \mathrm{loop}\, f\ a')(\mathrm{inr}\, b \Rightarrow b)$$

Dually:

$$\frac{f : A \times B \to A}{\mathrm{coloop}\, f : B \to A}$$

satisfying

$$\mathrm{coloop}\, f\ b = f\, \langle \mathrm{coloop}\, f\ b, b \rangle.$$

$\mathrm{coloop}\, f\ b$ is a fixed point of $\lambda a. f\, \langle a, b \rangle$.
So loops and fixed-points are dual. (Filinski 1994)

Inductive and coinductive data types

We want

- terminating functions
- pattern matching on data
- profit from duality

The categorical picture

Syntax for inductive data types is derived from categorical semantics:
Initial F-algebra: $(\mu F, \mathrm{in})$ s.t. $\forall (B, g)$, $\exists ! h$ such that the diagram commutes:

$$\begin{tikzcd}
F(\mu F) \arrow[r, "\mathrm{in}"] \arrow[d, "Fh"'] & \mu F \arrow[d, dashed, "!h"] \\
FB \arrow[r, "g"'] & B
\end{tikzcd}$$

- Due to the uniqueness: in is an isomorphism, so it has an inverse $\mathrm{out} : \mu F \to F(\mu F)$.
  In case $FX := 1 + X$, $\mu F = \mathrm{Nat}$ and out is basically the predecessor.

Inductive types are initial algebras

$$\begin{tikzcd}
F(\mu F) \arrow[r, "\mathrm{in}"] \arrow[d, "Fh"'] & \mu F \arrow[d, dashed, "!h"] \\
FB \arrow[r, "g"'] & B
\end{tikzcd}$$

- We derive the iteration scheme: a function definition principle + a reduction rule. The h in the diagram is called $\mathrm{It}\, g$

$$\frac{g : F\,B \to B}{\mathrm{It}\, g : \mu\, F \to B} \quad \text{with} \quad \mathrm{It}\, g\,(\mathrm{in}\, x) \;\twoheadrightarrow\; g\,(F(\mathrm{It}\, g)\, x)$$

- In case $FX := 1 + X$, $\mu F = \mathrm{Nat}$ and in decomposes in $0 : \mathrm{Nat}$, $\mathrm{Succ} : \mathrm{Nat} \to \mathrm{Nat}$;

$$\frac{d : D \quad f : D \to D}{\mathrm{It}\, d\, f : \mathrm{Nat} \to D} \quad \text{with} \quad \begin{array}{lcl} \mathrm{It}\, d\, f\, 0 & \twoheadrightarrow & d \\ \mathrm{It}\, d\, f\,(\mathrm{Succ}\, x) & \twoheadrightarrow & f\,(\mathrm{It}\, d\, f\, x) \end{array}$$

Primitive recursion

Given $d : B$, $g : \mathrm{Nat} \times B \to B$, I want $h : \mathrm{Nat} \to B$ satisfying

$$\begin{array}{lcl} h\,0 & \twoheadrightarrow & d \\ h(\mathrm{Succ}\,x) & \twoheadrightarrow & g\,x\,(h\,x) \end{array}$$

Defining primitive recursion

Given $d : B$, $g : \mathrm{Nat} \times B \to B$

$$
\begin{tikzcd}
1 + \mathrm{Nat} \arrow[r, "{[0, \mathrm{Succ}]}"] \arrow[d, "\mathrm{Id} + \langle h_1, h_2 \rangle"'] & \mathrm{Nat} \arrow[d, dashed, "!\langle h_1, h_2 \rangle"] \\
1 + (\mathrm{Nat} \times B) \arrow[r, "{[\langle 0, d \rangle, \langle \mathrm{Succ} \circ \pi_1, g \rangle]}"'] & \mathrm{Nat} \times B
\end{tikzcd}
$$

We derive the primitive recursion scheme:

- From uniqueness it follows that $h_1 = \mathrm{Id}$ (identity)
- From that we derive for h_2:

$$
\frac{d : B \quad g : \mathrm{Nat} \times B \to B}{h_2 : \mathrm{Nat} \to B} \quad \text{with} \quad
\begin{array}{lcl}
h_2\, 0 & \twoheadrightarrow & d \\
h_2(\mathrm{Succ}\, x) & \twoheadrightarrow & g\, x\, (h_2\, x)
\end{array}
$$

The induction proof principle also follows from this

Given $p_0 : P\,0$, $p_S : \forall x : \mathrm{Nat}. P\,x \to P\,(\mathrm{Succ}\,x)$

$$\begin{tikzcd}
1 + \mathrm{Nat} \arrow[r, "{[0, \mathrm{Succ}]}"] \arrow[d, "\mathrm{Id} + \langle h_1, h_2 \rangle"'] & \mathrm{Nat} \arrow[d, dashed, "!\langle h_1, h_2 \rangle"] \\
1 + (\Sigma x : \mathrm{Nat}. P\,x) \arrow[r, "{[\langle 0, p_0 \rangle, \mathrm{Succ} \times p_S]}"'] & \Sigma x : \mathrm{Nat}. P\,x
\end{tikzcd}$$

We derive the induction scheme:

- From uniqueness it follows that $h_1 = \mathrm{Id}$ (identity)
- From that we derive for h_2:

$$\frac{p_0 : P\,0 \qquad p_S : \forall x : \mathrm{Nat}. P\,x \to P\,(\mathrm{Succ}\,x)}{h_2 : \forall x : \mathrm{Nat}. P\,x}$$

In syntax, inductive types are only weakly initial algebras

$$\begin{tikzcd} F(\mu F) \arrow[r, "\mathrm{in}"] \arrow[d, "Fh"'] & \mu F \arrow[d, dashed, "h"] \\ FB \arrow[r, "g"'] & B \end{tikzcd}$$

- In syntax we only have weakly initial algebras: $\exists$, but not $\exists!$.
- So we get out and primitive recursion only in a weak slightly twisted form.
- We can derive the primitive recursion scheme via this diagram.

Primitive recursion scheme

Consider the following Primitive Recursion scheme for Nat. (Let D be any type.)

$$\frac{d : D \quad f : \mathrm{Nat} \to D \to D}{\mathrm{Rec}\, d\, f : \mathrm{Nat} \to D} \qquad \begin{array}{lcl} \mathrm{Rec}\, d\, f\, 0 & \twoheadrightarrow & d \\ \mathrm{Rec}\, d\, f\, (\mathrm{Succ}\, x) & \twoheadrightarrow & f\, x\, (\mathrm{Rec}\, d\, f\, x) \end{array}$$

One can define Rec in terms of It. (This is what Kleene found out at the dentist.)

$$\dfrac{\dfrac{\dfrac{d : D \qquad\qquad f : \mathrm{Nat} \to D \to D}{\langle 0, d\rangle : \mathrm{Nat} \times D \qquad \lambda z.\langle \mathrm{Succ}\, z_1, f\, z_1\, z_2\rangle : \mathrm{Nat} \times D \to \mathrm{Nat} \times D}}{\mathrm{It}\, \langle 0, d\rangle\, \lambda z.\langle \mathrm{Succ}\, z_1, f\, z_1\, z_2\rangle : \mathrm{Nat} \to \mathrm{Nat} \times D}}{\lambda p.(\mathrm{It}\, \langle 0, d\rangle\, \lambda z.\langle \mathrm{Succ}\, z_1, f\, z_1\, z_2\rangle\, p)_2 : \mathrm{Nat} \to D}$$

$\langle -, - \rangle$ denotes the pair; $(-)_1$ and $(-)_2$ denote projections.

Primitive recursion in terms of iteration

Problems:

- Only works for values. For the now definable predecessor P we have:

$$\begin{array}{rrcl} & P(\mathrm{Succ}^{n+1}\, 0) & \twoheadrightarrow & \mathrm{Succ}^{n}\, 0 \\ \text{but not} & P(\mathrm{Succ}\, x) & = & x \end{array}$$

- Computationally inefficient

$$P(\mathrm{Succ}^{n+1}\, 0) \twoheadrightarrow \mathrm{Succ}^{n}\, 0 \text{ in linear time}$$

Iterative, primitive recursive algebras, algebras with case

- An iterative T-algebra (also weakly initial T-algebra) is a triple $(A, \mathrm{in}, \mathrm{It})$
- An T-algebra with case is a triple $(A, \mathrm{in}, \mathrm{Case})$
- A primitive recursive T-algebra is a triple $(A, \mathrm{in}, \mathrm{Rec})$

$$\begin{tikzcd}
T\,A \arrow[r, "\mathrm{in}"] \arrow[d, "T(\mathrm{It}\ g)"'] \arrow[dr, phantom, "="] & A \arrow[d, dashed, "\mathrm{It}\ g"] \\
T\,B \arrow[r, "g"'] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \arrow[r, "\mathrm{in}"] \arrow[dr, "g"'] & A \arrow[d, dashed, "\mathrm{Case}\ g"] \\
 & B
\end{tikzcd}$$

$$\begin{tikzcd}
TA \arrow[r, "\mathrm{in}"] \arrow[d, "T\langle \mathrm{Rec}\ g, \mathrm{Id}\rangle"'] \arrow[dr, phantom, "="] & A \arrow[d, dashed, "\mathrm{Rec}\ g"] \\
T(B \times A) \arrow[r, "g"'] & B
\end{tikzcd}$$

Defining data types in lambda calculus

- Iterative algebras can be encoded as Church data types
- Algebras with case can be encoded as Scott data types
- Primitive recursive algebras can be encoded as Church-Scott or Parigot data types

Church numerals

The most well-known Church data type

$$\begin{array}{rcl} \overline{0} & := & \lambda x\, f.x \\ \overline{1} & := & \lambda x\, f.f\, x \\ \overline{2} & := & \lambda x\, f.f\,(f\, x) \end{array} \qquad \begin{array}{rcl} \overline{p} & := & \lambda x\, f.f^{p}\,(x) \\ \overline{\mathrm{Succ}} & := & \lambda n.\lambda x\, f.f\,(n\, x\, f) \end{array}$$

- The Church data types have iteration as basis. The numerals are iterators.
- Iteration scheme for Nat. (Let D be any type.)

$$\frac{d : D \quad f : D \to D}{\mathrm{It}\, d\, f : \mathrm{Nat} \to D} \quad \text{with} \quad \begin{array}{lcl} \mathrm{It}\, d\, f\, \overline{0} & \twoheadrightarrow & d \\ \mathrm{It}\, d\, f\, (\overline{\mathrm{Succ}}\, x) & \twoheadrightarrow & f\, (\mathrm{It}\, d\, f\, x) \end{array}$$

- Advantage: quite a bit of well-founded recursion for free.
- Disadvantage: no pattern matching built in; predecessor is hard to define. (Parigot: predecessor cannot be defined in constant time on Church numerals.)

Scott numerals

(First mentioned in Curry-Feys 1958)

$$\begin{array}{rcl} \underline{0} & := & \lambda x\, f.x \\ \underline{1} & := & \lambda x\, f.f\,\underline{0} \\ \underline{2} & := & \lambda x\, f.f\,\underline{1} \end{array} \qquad \begin{array}{rcl} \underline{n+1} & := & \lambda x\, f.f\,\underline{n} \\ \underline{\mathrm{Succ}} & := & \lambda p.\lambda x\, f.f\,p \end{array}$$

- The Scott numerals have case distinction as a basis: the numerals are case distinctors.
- Case scheme for Nat. (Let D be any type.)

$$\frac{d : D \quad f : \mathrm{Nat} \to D}{\mathrm{Case}\, d\, f\, : \mathrm{Nat} \to D} \quad \text{with} \quad \begin{array}{lcl} \mathrm{Case}\, d\, f\, \underline{0} & \twoheadrightarrow & d \\ \mathrm{Case}\, d\, f\, (\underline{\mathrm{Succ}}\, x) & \twoheadrightarrow & f\, x \end{array}$$

- Advantage: the predecessor can immediately be defined: $P := \lambda p.p\,\underline{0}\,(\lambda y.y)$.
- Disadvantage: No recursion (which one has to get from somewhere else, e.g. a fixed point-combinator).

Church-Scott numerals

Also called Parigot numerals (Parigot 1988, 1992).

$$
\begin{array}{lll|lll|lll}
 & & \text{Church} & & & \text{Scott} & & & \text{Church-Scott} \\
\overline{0} & := & \lambda x\, f.x & \underline{0} & := & \lambda x\, f.x & 0 & := & \lambda x\, f.x \\
\overline{1} & := & \lambda x\, f.f\; x & \underline{1} & := & \lambda x\, f.f\; \underline{0} & 1 & := & \lambda x\, f.f\; 0\, x \\
\overline{2} & := & \lambda x\, f.f\; (f\; x) & \underline{2} & := & \lambda x\, f.f\; \underline{1} & 2 & := & \lambda x\, f.f\; 1\, (f\; 0\, x)
\end{array}
$$

For Church-Scott:

$$
\begin{array}{rcl}
n+1 & := & \lambda x\, f.f\; n\, (n\, x\, f) \\
\mathrm{Succ} & := & \lambda p.\lambda x\, f.f\; p\, (p\, x\, f)
\end{array}
$$

Primitive recursion scheme for Nat. (Let D be any type.)

$$
\frac{d : D \quad f : \mathrm{Nat} \to D \to D}{\mathrm{Rec}\, d\, f : \mathrm{Nat} \to D} \quad \text{with} \quad
\begin{array}{lcl}
\mathrm{Rec}\, d\, f\; \underline{0} & \twoheadrightarrow & d \\
\mathrm{Rec}\, d\, f\; (\underline{\mathrm{Succ}}\, x) & \twoheadrightarrow & f\; x\, (\mathrm{Rec}\, d\, f\; x)
\end{array}
$$

Church-Scott numerals

Also called Parigot numerals (Parigot 1988, 1992).

		Church			Scott			Church-Scott
$\overline{0}$	$:=$	$\lambda x\, f.x$	$\underline{0}$	$:=$	$\lambda x\, f.x$	0	$:=$	$\lambda x\, f.x$
$\overline{1}$	$:=$	$\lambda x\, f.f\, x$	$\underline{1}$	$:=$	$\lambda x\, f.f\, \underline{0}$	1	$:=$	$\lambda x\, f.f\, 0\, x$
$\overline{2}$	$:=$	$\lambda x\, f.f\, (f\, x)$	$\underline{2}$	$:=$	$\lambda x\, f.f\, \underline{1}$	2	$:=$	$\lambda x\, f.f\, 1\, (f\, 0\, x)$

- Advantage: the predecessor can immediately be defined: $P := \lambda p.p\, \underline{0}\, (\lambda y.y)$.
- Advantage: quite a lot of recursion built in.
- Disadvantage: Data-representation of $n \in \mathbf{N}$ is exponential in n. (But: see recent work by Stump & Fu.)
- Disadvantage: No canonicity. There are closed terms of type Nat that do not represent a number, e.g. $\lambda x\, f.f\, 2\, x$.
  NB For Church numerals we have canonicity:
  If $\vdash t : \forall X.X \to (X \to X) \to X$, then $\exists n \in \mathbf{N}(t =_\beta \overline{n})$.
  Similarly for Scott numerals.

Typing Church and Scott data types

- Church data types can be typed in polymorphic λ-calculus, $\lambda 2$.
  E.g. for Church numbers: $\mathrm{Nat} := \forall X. X \to (X \to X) \to X$.
- To type Scott data types we need $\lambda 2\mu$: $\lambda 2$ + positive recursive types:
  - $\mu X.\Phi$ is well-formed if X occurs positively in Φ.
  - Equality is generated from $\mu X.\Phi = \Phi[\mu X.\Phi/X]$.
  - Additional derivation rule:

  $$\frac{\Gamma \vdash M : A \qquad A = B}{\Gamma \vdash M : B}$$

  For Scott numerals: $\mathrm{Nat} := \mu Y.\forall X. X \to (Y \to X) \to X$, i.e.

  $$\mathrm{Nat} = \forall X. X \to (\mathrm{Nat} \to X) \to X.$$

- Similarly for Church-Scott numerals:
  $\mathrm{Nat} := \mu Y.\forall X. X \to (Y \to X \to X) \to X$,

  $$\mathrm{Nat} = \forall X. X \to (\mathrm{Nat} \to X \to X) \to X.$$

Dually: coinductive types

Our pet example is Str_A, streams over A. Its (standard) definition in $\lambda 2$ as a "Church data type" is

$$\begin{array}{rcl} \mathrm{Str}_A & := & \exists X.X \times (X \to A \times X) \\ \mathrm{hd} & := & \lambda s.(s_2\, s_1)_1 \\ \mathrm{tl} & := & \lambda s.\langle (s_2\, s_1)_2, s_2 \rangle \end{array}$$

NB1: I do typing à la Curry, so $\exists$-elim/$\exists$-intro are done 'silently'.
NB2: $\langle -, - \rangle$ denotes pairing and $(-)_i$ denotes projection.
Two examples

$$\begin{array}{rcl} \mathrm{ones} & := & \langle 1, \lambda x.\langle 1, x \rangle \rangle : \mathrm{Str}_{\mathrm{Nat}} \\ \mathrm{nats} & := & \langle 0, \lambda x.\langle x, \mathrm{Succ}\, x \rangle \rangle : \mathrm{Str}_{\mathrm{Nat}} \end{array}$$

NB Representations of streams in λ-calculus are finite terms in normal form.

Constructor for streams?

Church data type Str_A

$$\begin{aligned}
\mathrm{Str}_A &:= \exists X. X \times (X \to A \times X) \\
\mathrm{hd} &:= \lambda s. (s_2\, s_1)_1 \\
\mathrm{tl} &:= \lambda s. \langle (s_2\, s_1)_2, s_2 \rangle
\end{aligned}$$

Problem: we cannot define

$$\mathrm{Cons} : A \to \mathrm{Str}_A \to \mathrm{Str}_A.$$

Problem arises because Str_A is only a weakly final co-algebra. (No uniqueness in the diagram.)
We need a co-algebra with co-case in the syntax or a primitive co-recursive co-algebra

Coinductive types are final co-algebra's

Final F-coalgebra: $(\nu F, \mathrm{out})$ s.t. $\forall (B, g)$, $\exists ! h$ such that the diagram commutes:

$$\begin{tikzcd}
B \arrow[r, "g"] \arrow[d, dashed, "!h"'] & FB \arrow[d, "Fh"] \\
\nu F \arrow[r, "\mathrm{out}"'] & F(\nu F)
\end{tikzcd}$$

For streams over A, $F\,X = A \times X$.

$$\begin{tikzcd}
B \arrow[r, "\langle g_1, g_2 \rangle"] \arrow[d, dashed, "!h"'] & A \times B \arrow[d, "\mathrm{Id} \times h"] \\
\mathrm{Str}_A \arrow[r, "\langle \mathrm{hd}, \mathrm{tl} \rangle"'] & A \times \mathrm{Str}_A
\end{tikzcd}$$

$$\begin{array}{rcl}
\mathrm{hd}(h\,b) & = & g_1\,b \\
\mathrm{tl}(h\,b) & = & h(g_2\,b)
\end{array}$$

Co-iterative, prim. co-recursive, co-algebras with co-case

- A co-iterative T-co-algebra (also weakly final T-co-algebra) is a triple $(A, \mathrm{out}, \mathrm{CoIt})$
- A T-co-algebra with co-case is a triple $(A, \mathrm{out}, \mathrm{CoCase})$
- A primitive co-recursive T-co-algebra is a triple $(A, \mathrm{out}, \mathrm{CoRec})$

$$\begin{tikzcd}
B \arrow[r, "g"] \arrow[d, dashed, "!\mathrm{CoIt}\ g"'] \arrow[dr, phantom, "="] & T\,B \arrow[d, "T(\mathrm{CoIt}\ g)"] \\
A \arrow[r, "\mathrm{out}"'] & T\,A
\end{tikzcd}
\qquad
\begin{tikzcd}
B \arrow[d, dashed, "\mathrm{CoCase}\ g"'] \arrow[dr, "g"] & \\
A \arrow[r, "\mathrm{out}"'] & T\,A
\end{tikzcd}$$

$$\begin{tikzcd}
B \arrow[r, "g"] \arrow[d, dashed, "\mathrm{CoRec}\, g"'] & T(A+B) \arrow[d, "T[\mathrm{Id}, \mathrm{CoRec}\, g]"] \\
A \arrow[r, "\mathrm{out}"'] & T\,A
\end{tikzcd}$$

For Streams over A this amounts to the following

Streams over A with CoCase and Streams over A with CoRec

$$\begin{tikzcd}
B \arrow[d, dashed, "\mathrm{CoCase}\ g"'] \arrow[dr, "g"] & \\
\mathrm{Str}_A \arrow[r, "\langle \mathrm{hd}, \mathrm{tl} \rangle"'] & A \times \mathrm{Str}_A
\end{tikzcd}
\qquad
\begin{tikzcd}
B \arrow[r, "g"] \arrow[d, dashed, "\mathrm{CoRec}\, g"] & A \times (\mathrm{Str}_A + B) \arrow[d, "\mathrm{Id} \times [\mathrm{Id}, \mathrm{CoRec}\, g]"] \\
\mathrm{Str}_A \arrow[r, "\langle \mathrm{hd}, \mathrm{tl} \rangle"'] & A \times \mathrm{Str}_A
\end{tikzcd}$$

- CoCase with $B := A \times \mathrm{Str}_A$ and $g := \mathrm{Id}$ gives the constructor for streams:

$$\mathrm{CoCase}\,\mathrm{Id} : A \times \mathrm{Str}_A \to \mathrm{Str}_A$$

- CoRec with $B := A \times \mathrm{Str}_A$ and $g := \mathrm{Id} \times \mathrm{inl}$ gives the constructor for streams:

$$\mathrm{CoRec}\,(\mathrm{Id} \times \mathrm{inl}) : A \times \mathrm{Str}_A \to \mathrm{Str}_A$$

Streams à la Scott and à la Church-Scott

Streams as a Church data type (in $\lambda 2$:

$$\mathrm{Str}_A \;:=\; \exists X.X \times (X \to A \times X)$$

Streams as a Scott data type (in $\lambda 2\mu$)

$$\begin{array}{rcll}
\mathrm{Str}_A & = & \exists X.X \times (X \to A \times \mathrm{Str}_A) & \\
\mathrm{hd} & := & \lambda s.(s_2\, s_1)_1 & \\
\mathrm{tl} & := & \lambda s.(s_2\, s_1)_2 & \\
\mathrm{Cons} & := & \lambda a\, s.\langle a, \lambda x.\langle a, s\rangle\rangle & [\text{take } X := A]
\end{array}$$

Streams as a Church-Scott data type (in $\lambda 2\mu$)

$$\begin{array}{rcll}
\mathrm{Str}_A & = & \exists X.X \times (X \to A \times (\mathrm{Str}_A + X)) & \\
\mathrm{hd} & := & \lambda s.(s_2\, s_1)_1 & \\
\mathrm{tl} & := & \lambda s.\mathrm{case}\,(s_2\, s_1)_2 \text{ of } (\mathrm{inl}\, y \Rightarrow y)\,(\mathrm{inr}\, x \Rightarrow \langle x, s_2\rangle) & \\
\mathrm{Cons} & := & \lambda a\, s.\langle a, \lambda x.\langle a, \mathrm{inl}\, s\rangle\rangle & [\text{take } X := A]
\end{array}$$

Streams à la Scott and à la Church-Scott

We immediately check that

$$\begin{array}{rcl} \mathrm{hd}(\mathrm{Cons}\, a\, s) & \twoheadrightarrow & a \\ \mathrm{tl}(\mathrm{Cons}\, a\, s) & \twoheadrightarrow & s \end{array}$$

Remark: Other definitions of Cons are possible, e.g.

$\mathrm{Cons} := \lambda a\, s. \langle \langle a, s \rangle, \lambda v. \langle v_1, \mathrm{inl}\, v_2 \rangle \rangle$ [take $X := A \times \mathrm{Str}_A$]

The general pattern (inductive types)

Let $\Phi(X)$ be a positive type scheme, i.e. X occurs only positively in the type expression $\Phi(X)$.

- We view $\Phi(X)$ as a functor on types. Positivity guarantees that Φ acts functorially on terms: we can define $\Phi(f)$ satisfying

$$\frac{f : A \to B}{\Phi(f) : \Phi(A) \to \Phi(B)}$$

- We can define an iterative Φ-algebra, a Φ-algebra with case and a primitive recursive Φ-algebra in the type theory as follows:
  - Church data type (iterative), in $\lambda 2$

$$A := \forall X.(\Phi(X) \to X) \to X$$

  - Scott data type (case), in $\lambda 2\mu$

$$A = \forall X.(\Phi(A) \to X) \to X$$

  - Church-Scott data type (primitive recursive), in $\lambda 2\mu$

$$A = \forall X.(\Phi(A \times X) \to X) \to X$$

The general pattern (coinductive types)

Let again $\Phi(X)$ be a positive type scheme.

We can define an co-iterative Φ-co-algebra, a Φ-co-algebra with co-case and a primitive co-recursive Φ-co-algebra in the type theory as follows:

- Church data type (co-iterative), in $\lambda 2$

$$A := \exists X. X \times (X \to \Phi(X))$$

- Scott data type (co-case), in $\lambda 2\mu$

$$A := \exists X. X \times (X \to \Phi(A))$$

- Church-Scott data type (primitive co-recursive), in $\lambda 2\mu$

$$A := \exists X. X \times (X \to \Phi(A + X))$$

Definition of streams in Coq

In the Coq system, CoInductive types are defined using constructors and not using destructors.
Question: Can we reconcile this?

```
CoInductive Stream (T: Type): Type :=
  Cons: T -> Stream T -> Stream T.
```

The destructors are defined by pattern matching.
How to define

```
    ones = 1 :: ones
```

with `ones : Stream nat`

```
CoFixpoint ones : Stream nat :=
  Cons 1 ones.
```

The recursive call to `ones` is guarded by the constructor `Cons`.
NB. The term `ones` does not reduce to `Cons 1 ones`.

Zipping and streams as sequences

The following definition is accepted by Coq

```
CoFixpoint zip (s t : Stream A) :=
 Cons (hd s) (zip t (tl s)).
```

There is an isomorphism between `Stream A` and `nat -> A`.

```
CoFixpoint F (f:nat->A) : Stream A :=
  Cons (f 0)(F (fun n:nat => f (S n))).
```

This defines

$$F(f) := f(0) :: F(\lambda n.f(n+1))$$

which is correct, because F is guarded by the constructor.

Deriving Coq's coinductive types from final coalgebras

$$\begin{tikzcd} B \arrow[r, "g"] \arrow[d, dashed, "!h"'] & FB \arrow[d, "Fh"] \\ \nu F \arrow[r, "\mathrm{out}"'] & F(\nu F) \end{tikzcd}$$

For a co-inductive type definition, Coq gives the following

- $\mathrm{Cons} : F\,(\nu\,F) \to \nu\,F$
- $\mathrm{out} \circ \mathrm{Cons} = \mathrm{Id}$
  (For streams: $\mathrm{hd}(\mathrm{Cons}\,a\,s) = a$ and $\mathrm{tl}(\mathrm{Cons}\,a\,s) = s$).
- $\forall x : \nu\,F, \exists y : F(\nu\,F), x = \mathrm{Cons}\,y$
- A guarded definition principle

Can we recover these from the final algebra diagram?

Coq's coinductive types from final coalgebras

$$
\begin{tikzcd}
B \arrow[r, "g"] \arrow[d, dashed, "!h"'] & FB \arrow[d, "F\,h"] \\
\color{red}{\nu F} \arrow[r, "\color{red}{\mathrm{out}}"'] & \color{red}{F(\nu F)}
\end{tikzcd}
$$

- We define $\mathrm{Cons} := \mathrm{CoIt}(F\,\mathrm{out})$ (the h we get if we take $g := F\,\mathrm{out}$.
- Then $\mathrm{out} \circ \mathrm{Cons} = \mathrm{Id}$ (By Lambek's Lemma)
- From this one can prove

$$\mathrm{Cons} \circ \mathrm{out} = \mathrm{Id}$$

- Then for $\forall x : A, \exists y : F\,A, x = \mathrm{Cons}\,y$, by taking $y := \mathrm{out}\,x$.

Deriving Coq's guarded definitions from final coalgebras for Str_A

$$
\begin{tikzcd}
B \arrow[r, "\langle g_1, g_2 \rangle"] \arrow[d, dashed, "!h"] & A \times B \arrow[d, "\mathrm{Id} \times h"] \\
\mathrm{Str}_A \arrow[r, "\langle \mathrm{hd}, \mathrm{tl} \rangle"'] & A \times \mathrm{Str}_A
\end{tikzcd}
$$

The left of the diagram can be further decomposed.

Coq's guarded definitions from final coalgebras

$$
\begin{tikzcd}
 & B \arrow[r, "\langle g_1, g_2 \rangle"] \arrow[dl, swap, "\langle g_1, h \circ g_2 \rangle"] \arrow[dd, dashed, "!h"] & A \times B \arrow[dd, "\mathrm{Id} \times h"] \\
A \times \mathrm{Str}_A \arrow[dr, swap, "\mathrm{Cons}"] & & \\
 & \mathrm{Str}_A \arrow[r, swap, "\langle \mathrm{hd}, \mathrm{tl} \rangle"] & A \times \mathrm{Str}_A
\end{tikzcd}
$$

Coq actually uses this property to define the function h.

CoFixpoint h (x:A) $:=$ $\mathrm{Cons}(g_1$ x) (h (g_2 x))

Programming with proofs

Following Krivine, Parigot, Leivant we can use proof terms in second order logic (AF2) as programs. This also works for recursively defined data types.
Assume some ambient domain U, with a constant $\mathbf{Z}$ and a unary function $\mathbf{S}$.
The natural numbers defined as a predicate on U:

$$\mathrm{Nat}(x) := \forall X.X(\mathbf{Z}) \to (\forall y.\mathrm{Nat}(y) \to X(y) \to X(\mathbf{S}\, y)) \to X(x)$$

When we erase all first order parts, we get the Church-Scott natural numbers:

$$\mathrm{Nat} := \forall X.X \to (\mathrm{Nat} \to X \to X) \to X$$

Programming with proofs

The method now defines the untyped λ-terms 0 and Succ as the proof-terms

$$\begin{array}{rcl} 0 & : & \mathrm{Nat}(\mathbf{Z}) \\ \mathrm{Succ} & : & \forall x.\mathrm{Nat}(x) \to \mathrm{Nat}(\mathbf{S}\,x) \end{array}$$

Then

$$\begin{array}{rcl} 0 & =_\beta & \lambda z\,f.z \\ \mathrm{Succ} & =_\beta & \lambda p.\lambda z\,f.f\;p\,(p\,z\,f) \end{array}$$

All the proofs of $\mathrm{Nat}(t)$ are representations of numbers; there is no 'junk'

Recursive programming with proofs

$$\mathrm{Nat}(x) := \forall X.X(\mathbf{Z}) \to (\forall y.\mathrm{Nat}(y) \to X(y) \to X(\mathbf{S}\,y) \to X(x)$$

Programming can now be done by adding a function symbol with an equational specification, e.g.

$$\begin{array}{rcl} \mathbf{A}(\mathbf{Z}, y) & = & y \\ \mathbf{A}(\mathbf{S}(x), y) & = & \mathbf{S}(\mathbf{A}(x, y)) \end{array}$$

Then give a proof term

$$\mathrm{Add} : \forall x, y.\mathrm{Nat}(x) \to \mathrm{Nat}(y) \to \mathrm{Nat}(\mathbf{A}(x, y))$$

The proof-term Add is an implementation of addition in untyped λ-calculus.

Corecursive programming with proofs

Given a data type A, and unary functions **hd** and **tl**, we define streams over A by

$$\mathrm{Str}_A(x) := \exists X.X(x) \times (\forall y.X(y) \to A(\mathbf{hd}\, y) \times X(\mathbf{tl}\, y))$$

We find that for our familiar functions hd and tl:

$$\begin{aligned} \mathrm{hd} := \lambda s.(s_2\, s_1)_1 &\quad : \quad \forall x.\mathrm{Str}_A(x) \to A(\mathbf{hd}\, x) \\ \mathrm{tl} := \lambda s.\langle (s_2\, s_1)_2, s_2 \rangle &\quad : \quad \forall x.\mathrm{Str}_A(x) \to \mathrm{Str}_A(\mathbf{tl}\, x) \end{aligned}$$

Adding equations for **ones**:

$$\begin{aligned} \mathbf{hd}(\mathbf{ones}) &= 1 \\ \mathbf{tl}(\mathbf{ones}) &= \mathbf{ones} \end{aligned}$$

we can give a proof term

$$\mathrm{ones} : \mathrm{Str}_{\mathrm{Nat}}(\mathbf{ones})$$

for example by taking

$$\mathrm{ones} := \langle \mathrm{Id}, \lambda x.\langle 1, \mathrm{Id} \rangle \rangle : \mathrm{Str}_{\mathrm{Nat}}$$

Correctness of corecursive programming with proofs

The proof term ones is guaranteed to be correct:

$$\begin{aligned}\mathrm{hd}(\mathrm{ones}) &\twoheadrightarrow 1 \\ \mathrm{tl}(\mathrm{ones}) &\twoheadrightarrow \mathrm{ones}\end{aligned}$$

Corecursive programming with proofs

To define Cons, we need to make Str_A into a recursive type:

$$\mathrm{Str}_A(x) := \exists X.X(x)\times(\forall y.X(y) \to A(\mathbf{hd}\, y)\times(\mathrm{Str}_A(\mathbf{tl}\, y)+X(\mathbf{tl}\, y)))$$

Adding equations for **Cons**:

$$\begin{array}{rcl} \mathbf{hd}(\mathbf{Cons}\, x\, y) & = & x \\ \mathbf{tl}(\mathbf{Cons}\, x\, y) & = & y \end{array}$$

We see that with

$$\mathrm{Cons} := \lambda a\, s.\langle\langle a, s\rangle, \lambda v.\langle v_1, \mathrm{inl}\, v_2\rangle\rangle$$

[take $X(x) := A(\mathbf{hd}\, x) \times \mathrm{Str}_A(\mathbf{tl}\, x)$].
we have

$$\mathrm{Cons} : \forall x, y, A(x) \to \mathrm{Str}_A(y) \to \mathrm{Str}_A(\mathbf{Cons}\, x\, y)$$

The typing system

To avoid syntactic overload and to get untyped λ terms, we use Curry style typing (as in AF2)

$$\frac{p : A \in \Gamma}{\Gamma \vdash p : A} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\,N : A} \qquad \frac{\Gamma, p : A \vdash M : B}{\Gamma \vdash \lambda p.M : A \to B}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall X.A}\, X \notin \mathrm{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall X.A}{\Gamma \vdash M : A[B(\vec{x})/X]}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall x.A}\, x \notin \mathrm{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall x.A}{\Gamma \vdash M : A[t/x]}$$

This works all very well for the inductive data types case

Problem

For the coinductive case, we have to deal with $\exists$. Curry-style exists rules are:

$$\frac{\Gamma \vdash M : A[B(\vec{x})/X]}{\Gamma \vdash M : \exists X.A}$$

$$\frac{\Gamma \vdash M : \exists X.A \quad \Gamma, p : A \vdash N : B}{\Gamma \vdash N[M/p] : B} \text{ if } X \notin \mathrm{FV}(\Gamma, B)$$

Problem: This system does not satisfy the not Subject Reduction property! (See counterexample in Sørensen-Urzyczyn)
The rules should be:

$$\frac{\Gamma \vdash M : A[B(\vec{x})/X]}{\Gamma \vdash \lambda h.h\,M : \exists X.A}$$

$$\frac{\Gamma \vdash M : \exists X.A \quad \Gamma, p : A \vdash N : B}{\Gamma \vdash M(\lambda p.N) : B} \text{ if } X \notin \mathrm{FV}(\Gamma, B)$$

Conclusion/Questions

- Church-Scott data types provide a good union of the two,
  - giving (co)-recursion in untyped λ-calculus
  - being typable in $\lambda 2 \mu$
  - but: the size of representation is a problem. (Recent work by Stump and Fu)
- We can prevent closed terms that don't represent data, by moving to types in AF2

Some questions:

- Can the "programming with proofs" approach in AF2 for inductive types fully generalize to coinductive types? Using Curry-style typing?
- Does that include corecursive types?
- Can we reconcile with the "naive" looping intuition?

Related Work on (co)inductive types in non-dependent type theories, lots

- Mendler style inductive/coinductive types: Mendler, Matthes, Uustalu, Vene
- Extending to course-of-value recursion: Matthes, Uustalu, Vene
- Impossibility results: Parigot, Malaria, Splawski & Urzyczyn
- General recursion via coinductive types: Capretta
- Recursive Coalgebras and Corecursive Algebras: Osius; Capretta & Uustalu & Vene

Related Work on coinductive types in dependent type theories

- Coquand, Gimenez
- Copatterns by Abel, Pientka, Setzer
- Type theory based solely on inductive and coinductive types: Basold, H.G.

Related Work on programs from proofs, lots

- Mendler style inductive/coinductive types: Miranda-Perea & González-Huesca
- Christophe Raffalli: infinitary terms
- Tatsuta: first order logic with (co)-inductive definitions
- Leivant

Related Work on Scott numerals/data

- "Types for Scott Numerals" Abadi, Cardelli, Plotkin 1993
- Brunel & Terui: capture polynomial time functions using Scott data types and linear types.
- Similar use in Baillot & De Benedetti & Ronchi della Rocca
- Scott data types with call-by-value and call-by-name iteration (H.G.)

Questions?